algorithms

MDPI

Article

Programming Agents by Their Social Relationships: A Commitment-Based Approach

Matteo Baldoni †,‡,*, **Cristina Baroglio** ‡, **Roberto Micalizio** ‡ **and Stefano Tedeschi** ‡

Dipartimento di Informatica, Università degli Studi di Torino, Torino 10149, Italy; baroglio@di.unito.it (C.B.); roberto.micalizio@unito.it (R.M.); stefano.tedeschi@unito.it (S.T.)

* Correspondence: baldoni@di.unito.it; Tel.: +39-011-670-6756

† Current address: via Pessinetto 12, 10149 Torino, Italy.

‡ These authors contributed equally to this work.

Received: 19 March 2019; Accepted: 13 April 2019; Published: 16 April 2019

Abstract: Multiagent systems can be seen as an approach to software engineering for the design and development of complex, distributed software. Generally speaking, multiagent systems provide two main abstractions for modularizing the software: the agents and the environment where agents operate. In this paper, we argue that also the social relationships among the agents should be expressed explicitly and become first-class objects both at design- and at development-time. In particular, we propose to represent social relationships as commitments that are reified as resources in the agents' environment and can be directly manipulated by the agents via standard operations. We demonstrate that this view induces an agent programming schema that is substantially independent of the actual agent platform, provided that commitments are available as explained. The paper exemplifies the schema on two agent platforms, JADE and JaCaMo, where commitments are made available via the 2COMM library.

Keywords: social commitments; agents and artifacts; agent-oriented software engineering

1. Introduction and Motivation

Multiagent Systems (MAS) are a preferred choice for building complex systems where the autonomy of each component is a major requirement. Agent-oriented software engineers can choose from a substantial number of agent platforms (see, e.g., [1] for an overview). Tools like JADE [2], TuCSoN [3], DESIRE [4], and JaCaMo [5] all provide coordination mechanisms and communication infrastructures, but in our opinion, they lack abstractions that allow a clear and explicit modeling of interaction. Basically, the way in which agents interact is spread across and "hard-coded" into agent implementations. This choice overly ties agent implementations with a negative impact on software reuse. A clear separation of the agents from the specification of their coordination would bring advantages both on the design and on the implementation of MAS by bringing in a greater decoupling. Our claim is that an explicit representation of the *social relationships* among agents is beneficial since it improves code modularity and flexibility in the interaction. We practically demonstrate these advantages when social relationships are modeled as *social commitments* [6] and *reified* as resources in the environment. A commitment is a promise, or contract, that an agent (debtor) makes to another one (creditor). Commitments have a normative power since they induce an obligation upon the debtor to bring about a condition, when a specific context holds true. A peculiarity of commitments is that they can only be created by their debtor. Thus, by properly selecting the terms of the contract, an agent can create a commitment so as to entice the cooperation of another agent.

This paper significantly extends and is complementary to the work presented in [7]. The main focus, here, is on the definition of a *conceptual architecture* for modeling multiagent interactions, which encompasses the notion of social relationship as a first-class entity. Social relationships,

captured by social commitments, are made available to the agents as artifacts, that the agents can directly manipulate instead of passing through interaction protocols. Despite relying on the 2COMM infrastructure, which is used here to exemplify the feasibility of the approach, the discussion is abstract and aims at providing a general model that can be used by the systems' designers to program agent interaction. As distinct from previous work, like [8], the attention is placed on the characterization of the key components of the conceptual architecture and on how they can provide a guideline for the development of the individual agents. It is not focused on the programming aspects related to the development of an infrastructure, supporting the picture above (as the previous work was).

One of the paper's major contributions lies in the definition of a schema, rooted in the mentioned architecture, for programming the way in which agents create and manage their interactions with others. We show how the modeling of interaction is a good starting point in the development of a distributed system when it comes to programming the agents. To this end, we rely on social commitments [9]. In particular, we show how, by relying on reified commitments, it is possible to devise schemas for programming agents based upon handling the agents' mutual engagements. Notably, such schemas are independent of any specific agent platform because they only depend on the standardized lifecycle of commitments. Two programming schemas are introduced for helping the programmer: *entice* (used to obtain the cooperation of another agent by creating commitments) and *cooperate* (used to properly handle commitment state changes). The 2COMM [8] tool is used to exemplify the implementation of such a platform by building artifacts that incorporate commitments in a way that is not bound to any agent platform. Indeed, for using 2COMM within a specific agent platform, it is sufficient to realize a dedicated connector. So far, two connectors make 2COMM available to JADE and JaCaMo agents.

Relying on artifacts has the advantage of transforming social relationships and coordination schemas into *resources*, and this allows agents to recognize, accept, manipulate, reason on them, and decide whether to conform to them dynamically (a basis for coordination [10]). In order to turn the social relationships into reified resources, we rely on the Agents and Artifacts meta-model (A&A) [11,12], which provides abstractions for environments and artifacts that can be acted upon, observed, perceived, notified, and so on. 2COMM adopts the abstraction of artifact to construct commitment artifacts that realize a form of mediated, programmable communication.

Finally, we resort to a novel logistics scenario to explain the practical use of the two patters, showing how the interactions among a number of agents can be implemented in a uniform way both for JADE and for JaCaMo agents.

Organization

The paper is organized as follows. After a brief positioning regarding the state-of-the-art and a short recall about social commitments, reported in Section 3, Section 4 presents the conceptual architecture for an agent platform supporting commitments and general programming schemas for agents, grounded in the described architecture. The 2COMM middleware is introduced as a possible practical realization of the conceptual architecture in Section 5, where the general programming schema is applied in two existing agent platforms: JADE and JaCaMo. Section 6 exemplifies the agent programming schema in a non-trivial logistic scenario in both agent platforms used as a reference.

2. Related Works

Coordination is a critical aspect of multiagent systems, and not surprisingly, many initiatives exist in the literature that aim at modeling it (see, e.g., [13,14]). A fundamental approach for modeling coordination is based on message protocols. For instance, the well-known Contract Net Protocol (CNP) [15] has been developed to specify problem-solving communication and control for a group of individuals in a distributed setting. A disadvantage of message protocols, however, is that the interaction logic is intermingled with the agent control logic. This lack of concern separation hampers software modularity and reuse. As advocated by Philippsen [16] in his survey on Concurrent

Object-Oriented Languages (COOLs), for promoting software modularity, coordination should be implemented *on the side of the callee*. Namely, the coordination should be implemented outside the agents, in a class/resource that is accessed concurrently by the agents being coordinated.

This limitation of message protocols has led many researchers in the field to look at agent organizations as a promising way to model coordination. Intuitively, an organization establishes a society of agents that is characterized by a set of organizational goals and a set of norms. Agents, playing one or more roles, should accomplish the societal goals respecting the norms. Electronic institutions [17,18], similarly to organizations, use norms for regulating the agents' interactions. Differently from agents, however, electronic institutions have no goals of their own and can be considered as a sort of monitor controlling the agents' behaviors. For example, the abstract architecture of e-institutions envisioned by Ameli [19,20] places a middleware, made of governors and staff agents, between participating agents and an agent communication infrastructure. The environment is nothing agents can sense and act upon, but rather, it is a conceptual one. Agents communicate with each other by means of speech acts, and behind the scene, the middleware mediates such communication thanks to a body of norms and laws.

Organizations add to the picture the notion of organizational goal, distributing it through the roles that make up the organization [21]. Among the current proposals, the organizational infrastructure in [22], as well as JaCaMo are based on *Moise*$^+$, which allows both for the enforcement and the regimentation of the rules of the organization. This is done by defining a set of conditions to be achieved and the roles that are permitted or obliged to perform them. Nevertheless, as already pointed out in [23], very few agent frameworks address *interaction* as a first-class entity. An exception is represented by the *interaction component* in [24,25] for the JaCaMo platform. The interaction component enables both agent-to-agent and agent-to-environment interaction, providing guidelines of how a given organizational goal should be achieved, with a mapping from organizational roles to interaction roles. Guidelines are encoded in an *automaton-like* shape, where states represent protocol steps, and transitions between states are associated with (undirected) obligations: the execution of such steps creates *obligations* on some agents in the system, which can concern actions performed by the agents in the environment, messages that an agent sends to another agent, and events that an agent can perceive (i.e., events emitted from objects in the environment). The specification of interaction via automata, however, shows a rigidity that prevents agents from taking advantage of the opportunities and of handling exceptions in dynamic and uncertain multiagent environments [26,27]. Agents are, in fact, confined to the execution sequences provided by the automaton.

In contrast to this approach, since the seminal papers by Yolum and Singh [26,28], commitment-based protocols have been raising much attention (see, e.g., [29,30]). Protocol actions affect the state of the system, which consists both of the state of the world and also of the *commitments* that agents have made to each other. Commitments motivate agents to perform their next actions. This happens because agents want to comply with the protocol and provide what was promised to the other parties. Another key feature of commitments is their *declarative nature*, which easily accommodates the contractual relationships among the partners rather than strictly encoding the order in which messages should be exchanged. Whatever action agents decide to perform is acceptable as far as they accomplish their commitments, satisfying the expectations they have on one another. The JaCaMo+ proposal [23] relies on commitments to model explicitly the relationships among the agents as first-class elements of an agent organization. In this paper, we build upon the experience of JaCaMo+ and discuss how the commitment-based approach can be made independent of any agent platform.

3. Social Relationships as Commitments

We propose to program the interaction between agents by explicitly representing the social relationships among them. In our view, social relationships should become resources that are reified as part of the environment; they should also be subject to *social control*, thus acquiring a normative and regulative value. For this reason, we rely on the notion of *social commitment*. A social commitment [9],

represented as $\mathsf{C}(x, y, p, q)$, captures the fact that an agent x (debtor) commits towards an agent y (creditor) to bring about a consequent condition q when an antecedent condition p will hold. Antecedent and consequent conditions are generally expressed in *precedence logic* [31], an event-based linear temporal logic for modeling web services' composition. It deals with occurrences of events along runs (i.e., sequence of instanced events). Event occurrences are assumed to be non-repeating and persistent: once an event has occurred, it has occurred for the whole execution. The logic has three operators: "$\vee$" (choice), "$\wedge$" (co-occurrence), and "$\cdot$" (before). The *before* operator allows constraining the order with which two events must occur; e.g., $a \cdot b$ means that a must occur before b, but the two events do not need to occur one immediately after the other.

Commitments are proactively created by debtors, manifesting the agent capacity to take responsibilities towards bringing about some conditions autonomously. Moreover, commitments have a standardized lifecycle, formalized in [32] and summarized in Figure 1. As soon as a commitment is created, it is *active*. Active commitments can further be in two sub-states: *conditional* when neither the antecedent, nor the consequent have occurred, and *detached* as soon as the antecedent becomes true. A commitment is *violated* either when its antecedent is true, but its consequent becomes false or when it is canceled by the debtor when detached. Conversely, it is *satisfied* when the consequent occurs and the engagement is accomplished. It is *expired* when it is no longer in effect; i.e., if the antecedent condition becomes false. Finally, a commitment becomes *terminated* when the creditor releases the debtor from it or when it is canceled, being still conditional. Social commitments can be manipulated by the agents through some standard operations: namely, *create*, *cancel*, *release*, *discharge*, *assign*, and *delegate* [9]. *Create* instantiates a commitment, changing its status from *null* to *active*. Only the debtor of a commitment can create it. Conversely, *cancel* revokes the commitment, setting its status to *terminated* or *violated*, depending on whether the previous status was *conditional* or *detached*, respectively. Again, only the debtor can cancel a commitment. The rationale is that a commitment debtor is allowed to change its mind about the accomplishment of the consequent only until the antecedent has not occurred, yet. *Release*, in turn, allows the creditor (and only the creditor) to terminate an active commitment. Release does not mean success or failure of the given commitment, but simply eliminates the expectation put on the debtor. *Discharge* satisfies the commitment. In accordance with [9], we assume that this operation is performed concurrently with the actions that lead to the consequent occurrence. *Delegate* changes the debtor of a given commitment and can be performed only by the new debtor. *Assign*, finally, transfers the commitment to another creditor and can be performed only by the actual one. An active commitment can be further suspended, i.e., put in the *pending* status, by means of the *suspend* operation. Conversely, *reactivate* sets a pending commitment as active, again.

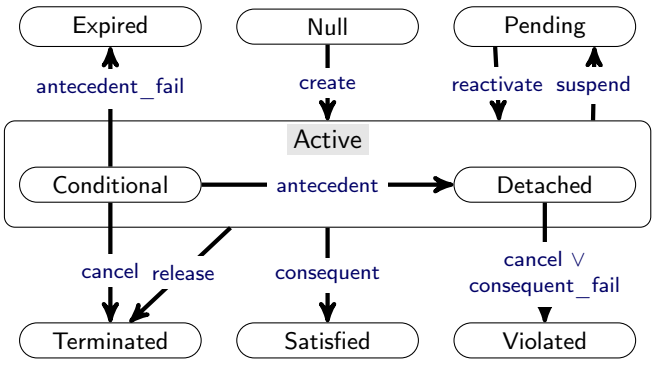

Figure 1. Commitment life cycle [32].

Since a commitment can only be taken autonomously by a debtor towards a creditor on its own initiative, the agents' autonomy is preserved, and this aspect is fundamental in order to harmonize deliberation with goal achievement. An agent will create engagements towards other agents in an attempt to achieve its goals. Moreover, since commitments concern the observable behavior of the

agents, they acquire a regulative value: debtors are expected to satisfy their engagements in order to avoid a norm violation. More concretely, commitments foster cooperation between agents. On the one hand, debtors are expected to behave so as to satisfy their detached commitments. On the other hand, a creditor in a commitment that is interested in its consequent condition, will act so as to bring about the antecedent condition, thus detaching the very same commitment, and by this very act claim, the consequent achievement. In this sense, commitments allow realizing a *relational representation of interaction*, where agents, by their own actions, directly create (normative) binds with others and use them to coordinate their activities. Commitment-based approaches generally assume a *social state* to be available and inspectable by all the agents involved in an interaction. Such a social state is composed of all the commitments created during an interaction together with their states, according to the life-cycle described above. By relying on the social state, an agent can deliberate to create further commitments or to bring about a condition involved in some existing one.

By means of commitments, thus, social relationships become first-class entities that are created and manipulated by the agents as resources, made available in their environment. It is, therefore, necessary to provide the agents the means to create, manipulate, observe, monitor, reason, and deliberate upon these social relationships. To this end, we rely on the well-known agents and artifacts meta-model [11,12]. Following this perspective, an artifact is a computational, programmable system resource that can be observed and manipulated by agents, residing at the same abstraction level. Artifacts are typically used to model the environment in which agents are situated and on which they act. Furthermore, for their very nature, artifacts can encode a programmable and observable middleware that can be used as a means of communication and coordination between agents.

We propose to define *dedicated commitment artifacts*, embodied in the environment in which agents are situated, to reify the sets of *social relationships* that can be created during an interaction, available to agents as resources. Agents can then use these artifacts to coordinate and interact in a way that depends on their objectives and on the binds created with other agents. We interpret the usage of an artifact by an agent as the explicit acceptance of the implications of the social commitment the artifact reifies. This allows the interacting parties to perform *practical reasoning*, based on social expectations: a debtor of a commitment is expected to behave so as to satisfy the consequent commitment conditions; otherwise, a violation will be raised. Similarly, an agent could create a commitment to foster another agent to bring about a condition otherwise unachievable.

Example 1. A Logistics Scenario

We present a use case inspired by a logistic scenario to show how commitments can be used in practice to support the interaction between agents. Let us consider a setting in which a *seller* agent sells its products online and ships them to a *customer* agent. In our scenario, sketched in Figure 2, the two agents, *seller* and *customer*, are geographically far from each other, so they need to rely on multiple couriers for the shipment of the goods from their original location A to their destination D. The route is divided into three parts. The first portion, from A to B, is covered by two different trucks $trk1$ and $trk2$. The second portion, from B to C, instead, is covered only by a plane $pln1$. Finally, the last portion, from C to D, is served again by two trucks $trk3$ and $trk4$. A first way to model the relationships among all these agents is to consider the following set of commitments.

$$\boldsymbol{c_1} : \mathsf{C}(seller,\ customer,\ pay(500, seller),\ at(goods, D))$$
$$\boldsymbol{c_2} : \mathsf{C}(trk1,\ seller,\ pay(50, trk1) \wedge at(goods, A),\ at(goods, B))$$
$$\boldsymbol{c_3} : \mathsf{C}(pln1,\ seller,\ pay(200, pln1) \wedge at(goods, B),\ at(goods, D))$$
$$\boldsymbol{c_4} : \mathsf{C}(trk3,\ pln1,\ pay(50,\ trk3) \wedge at(goods, C),\ at(goods, D))$$

Commitment c_1 represents the offer that *seller* proposes to *customer*: if *customer* pays 500 Euros for some *goods*, *seller* will deliver the bought goods at *customer*'s place D. Since *seller* cannot directly deliver the goods, however, it creates commitment c_1 only after having inspected the environment (i.e., the social state) and having observed the commitments c_2 and c_3. With the former, the *seller* gets

a means for moving the goods from A to B by using $trk1$. With the latter, instead, the *seller* has an agreement with $pln1$ to ship goods from B to D. In these two commitments, the *seller* plays the creditor role, and hence, it can detach them (by paying the asked amount) only after the *customer* has detached c_1 by paying for the goods. The last commitment, c_4, encodes an agreement between $pln1$ and $trk3$ for the shipping of goods from C to D, but the *seller* does not need to know how $pln1$ will satisfy c_3. From the *seller*'s perspective, in fact, it is sufficient to know that $pln1$ has taken on a commitment to ship the goods directly to the *customer*. Of course, $pln1$ creates c_3 only after c_4 has been created.

This scenario, despite being very simple, encompasses several alternative solutions, each consisting of a different set of commitments that agents create to entice the others. To keep the discussion simple, here we introduce a first, plain solution, whereas in Section 6, we will discuss alternative execution paths and demonstrate the flexibility we get when commitments are available as resources.

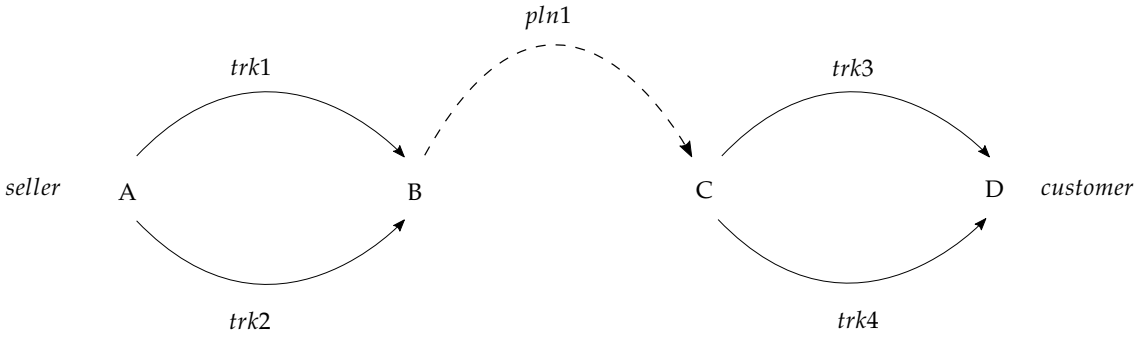

Figure 2. The logistic scenario.

4. A Conceptual Schema for Programming Agents with Social Relationships

We believe that social relationships should be considered as first-class entities of any agent programming platform. When social relationships are directly available as programming constructs to agents and programmers, in fact, the resulting system enjoys some important software engineering properties. First, it is possible to standardize the agent programming activity by following a general schema that is independent of the specific agent platform at hand. In this section, we focus on the general schema, while its practical use is exemplified in the next sections. In particular, we focus on social relationships that can be seen as commitments and reify them as computational resources that can be directly used by agents.

The conceptual architecture we propose is shown in Figure 3 where, to emphasize the generality of the programming schema, we abstract from any actual implementation and hence represent only classes without denoting fields and methods. The architecture encompasses both the classes that regard the agent being implemented (in yellow) and the classes through which commitments are made available (in blue). The rationale is that the programmer should be able to implement the agent code by only implementing the yellow classes, using the objects provided by the underlying commitment platform (in blue).

We first explain the classes and relations concerning the commitment platform. `Workspace` abstracts the part of environment that is of interest for the agent at hand. The workspace will collect relevant pieces of information and resources such as events (see `Event`) and commitments (see `Commitment`). In particular, some events represent the occurrence of commitment state changes—for instance, the satisfaction of a commitment or its detachment—and, thus, they are relevant for some agent in the system. Such events belong to the class *social* (see the specialization of `Event` by `Social Event`). Notice that an actual implementation of `Workspace` could maintain references of various types of resources, but in this conceptual architecture, we emphasize only those domain-independent classes that are essential for the reification of commitments and for the implementation of the agents.

The class `Commitment` abstracts the reification of the social relationships into a resource that can be directly manipulated by the agents. In fact, by inspecting the `Workspace`, the agents can determine the set of possible commitments together with their current states. Agents who are debtors or creditors of a commitment can also operate upon them directly through commitment operations. This is made possible by the class `Role`, whose instances amount either to the debtors or the creditors of commitment instances. Commitments are also characterized by antecedent and consequent conditions, which are both abstracted by class `Condition`. Note that `Condition` is in relation to `Event` because the satisfaction or violation of a condition is seen as an event occurring in the workspace. In particular, thanks to the specialization `Social Event`, it is possible to specify commitments, whose antecedent or consequent conditions include operations on commitments. This allows the specification of nested commitments as commitments whose antecedent or consequent conditions amount to the creation of another commitment. All the explained classes represent the concepts that should be implemented by a platform providing social commitments as programming constructs. Their actual implementation, however, could require much more concrete classes. As an exemplification, in the following section, we will sketch the specific case of the 2COMM library, and we will show how the very same platform can be interfaced with the JADE and with the JaCaMo agent programming platforms.

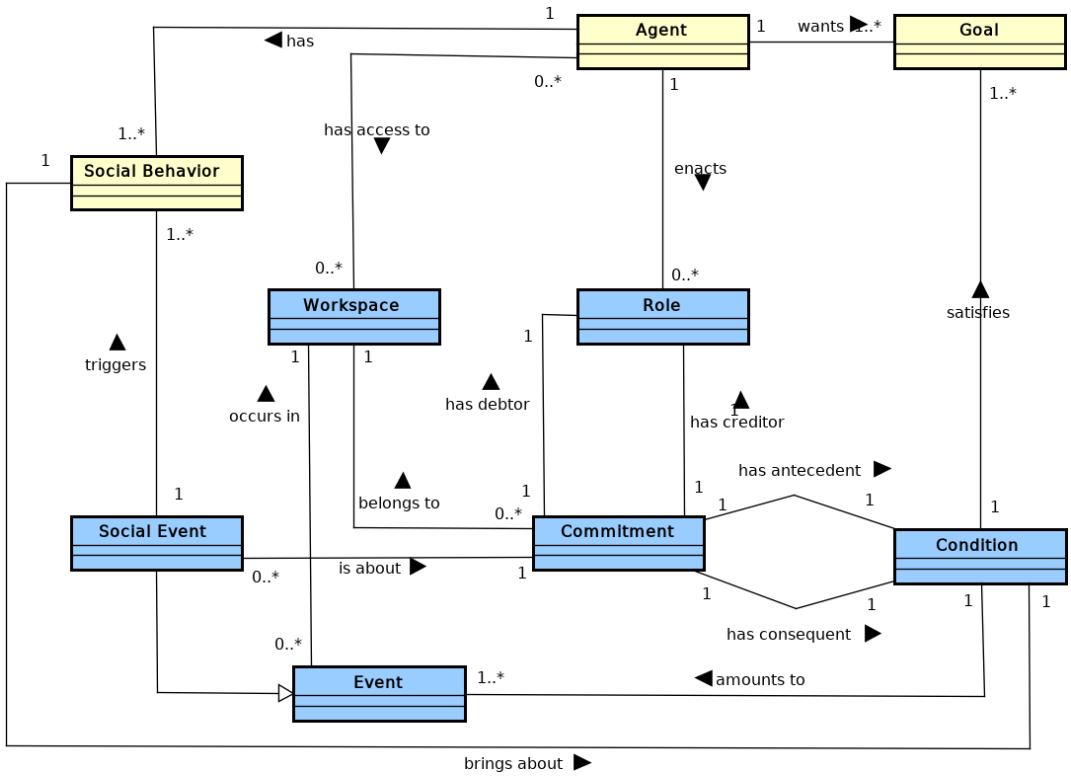

Figure 3. The conceptual architecture of agents and their relationships.

Concerning the agent classes (in yellow in Figure 3), an `Agent` has at least one `Goal` that it wants to achieve, and it is equipped with at least one `Social Behavior`. This is a behavior that brings about a `Condition` that occurs as antecedent or consequent of some commitment; the resulting commitment state change amounts to a `Social Event`, which may, in turn, activate other social behaviors. Agents can also access `Workspaces` by means of the roles they play. This feature is exploited to allow agents to obtain the cooperation of other agents for pursuing their goals. The process relies on the use of commitments as a means to express promises the help seeker will make to those accepting to help it.

4.1. Programming Schema for Help-Seeking Agents

The need for cooperation justifies a general programming schema for agents that use social relationships (i.e., commitments), for having their goals satisfied by other agents. The aim of the schema is, thus, to implement one (or more) social behaviors that allows an agent ag to achieve a goal G by cooperating with other agents. The schema consists of the following cases:

1. *Entice*:

   - *Intent*: Finding the cooperation of another agent by making an offer.
   - *Applicability*: When, by inspecting the workspace, the agent does not discover cooperation offers by other agents that support the achievement of its goal adequately, then the agent creates an offer itself.
   - *Implementation*: Implement a behavior that creates a commitment $\mathsf{C}(ag, other, G, q)$, where ag is the help-seeking agent, $other$ is another agent in the system, and q is a condition that ag can bring about, either directly or via the cooperation with others.

2. *Cooperate*:

   - *Intent*: contributing to a collaboration by making a commitment progress towards satisfaction.
   - *Applicability*: when the agent is either the debtor or the creditor of an active commitment.
   - *Implementation*:
     - When ag is the creditor: implement a behavior that is triggered by the creation of a commitment $c : \mathsf{C}(other, ag, p, G)$ and that brings about p, so as to detach c.
     - When ag is the debtor: implement a behavior that is triggered by the detachment of a commitment $c : \mathsf{C}(ag, other, G, q)$ and that brings about q so as to satisfy c.

Note that the *Cooperate* schema could be generalized to capture the whole standard lifecycle of a commitment. For instance, a creditor agent could also implement a behavior for releasing a commitment or when a commitment is violated, but these are choices that are driven by a local decision strategy of the agent.

Example 2. A Logistics Scenario (Continued)

Let us explain how the schema is applied to the previously-introduced logistic scenario. For the purpose of generality, we use an abstract language based on ECA rules (Event, Condition, Action), for expressing the agents' behaviors. Intuitively, an ECA rule has the following syntax: $event : condition \leftarrow action$. The event denotes the trigger for activating such a rule; in general, it is a goal that the agent wants to achieve or an event that must be properly treated by the agent. The condition is a contextual circumstance that must hold for the rule to be actually fired. Finally, the action is a course of operations that modify the environment so as to obtain a desired effect. In Section 4, we will show how these abstract behaviors can be implemented in the JADE and JaCaMo platforms.

Let us take the *seller*'s perspective, who wants to get some money by selling goods. It can do so by exploiting the *Entice* schema, in such a way to create a commitment $c_1 : \mathsf{C}(seller, customer, pay(500, seller), at(goods, D))$ to make an offer to a potential customer. This could be captured by the ECA rule:

$ON\ need\_money\ IF\ have\_goods\_to\_sell$
$DO\ \texttt{create}\ (c_1 : \mathsf{C}(seller, customer, pay(500, seller), at(goods, D)))$

The use of the entice schema is strictly connected to the use of the *Cooperate* one, which allows tackling the relevant commitment state changes. Let us assume that *seller* is only interested in a change from conditional to detached. The ECA rule would be:

ON `detached`(c_1) *IF true DO bring_goods_at_D*.

The triggering event is the detachment of commitment c_1, that is *customer* has paid for the goods. The contextual condition is true, meaning that the agent is always eager to react to such an event (but domain-dependent conditions could be specified). The action *bring_goods_at_D* stands for a plan for delivering the goods to the customer. That is, the effect of the plan would be $at(goods, D)$, which satisfies the commitment. Such a plan is not trivial since it requires the cooperation of other agents, that is modeled by commitments c_2, c_3, and c_4 that are assumed to exist already.

Let us now take the *customer*'s perspective, who has two choices. First, the customer can accept an offer by detaching the conditional commitment c_1. Thus, *customer* applies the *Cooperate* schema as creditor and detaches an existing commitment; the ECA rule is as follows:

ON `create`$(c_1 : \mathsf{C}(seller, customer, pay(500, seller), at(goods, D)))$
IF have_enough_money DO $pay(500, seller)$.

Namely, the triggering event is the creation of commitment c_1; the condition is having enough money for the goods; while the action is the payment, which will detach the commitment.

The second option of *customer* is to make a counteroffer to the seller by creating a new commitment. In such a case, *customer* will apply first the *Entice* schema and then the *Cooperate* one as the debtor similarly to the implementation of the seller. This second alternative available to the customer is detailed in Section 6 together with a sketch of the implementation of the Jadeand JaCaMo agents.

5. 2COMM: An Infrastructure for Programming Social Relationships

We have claimed that an agent-based framework should provide an explicit representation of the social relationships created between the agents as first-class entities and that these relationships should be used directly for programming the agent behaviors. 2COMM [8] is a middleware that fills such a gap by reifying social commitments as artifacts available in the environment. Currently, 2COMM supports social relationship-based agent programming for JADE and JaCaMo agents through two dedicated connectors. JADE [2] is an agent platform that supplies standard agent services, i.e., message passing, distributed containers, naming and yellow pages services, and agent mobility. JaCaMo [5], in turn, is a platform integrating three other frameworks: Jason [33] for programming agents, CArtAgO [34] for programming artifact-based environments, and Moise [35] as a support to the realization of multiagent organizations. The loose coupling between the agent-programming dimension and the interaction component makes the development of new connectors for other agent platforms straightforward.

2COMM relies on CArtAgO for the implementation of a (distributed) social state. Specifically, CArtAgO provides a way to define and organize *workspaces*, which are logical groups of artifacts, which can be joined by agents at runtime. The CArtAgO API allows programming artifacts, regardless of the agent programming language or the agent framework used. This is possible by means of the *agent body* metaphor: CArtAgO provides a native agent entity, which acts as a "proxy" in the artifacts workspace. An agent developed in a given agent programming platform is the "mind" that uses the CArtAgO agent entity as a "body" to interact with artifacts. An agent, in particular, acts upon an artifact by executing public *operations*, which can be equipped with *guards*; i.e., conditions that must hold for operations in order to produce their effects. An artifact provides also some *observable properties* that can be perceived by the agents *focusing* on it and encode the artifact's current state. Observable properties can change over time as a result of the operations occurring inside the artifact.

By means of CArtAgO, 2COMM reifies social commitments as a special class of *commitment artifacts*. Each commitment artifact provides two roles that agents can enact, corresponding to the debtor and to the creditor of the commitment maintained by the artifact. Roles can be seen as proxies that allow agents to operate on the corresponding commitment artifact. For instance, by adopting the debtor role, an agent will be able to perform the artifact operations to create the corresponding commitment or to cancel it. Roles are linked to agents of the specific platform via dedicated connector classes. 2COMM

provides the classes for representing commitments, maintained into commitment artifacts, as well as roles. Moreover, the infrastructure automatically handles the commitment progressions according to the events occurring in the environment. In particular, events that are relevant for the progression of commitments are encoded as *social facts*. 2COMM maintains a trace of all the social facts asserted in a given interaction and updates the involved commitments accordingly.

Agent programming with 2COMM amounts, at its core, to realizing a classical "sense-plan-act cycle", whose phases can be renamed "observe the workspace", "activate behaviors according to the entice/cooperate schemas", and "schedule behavior execution". In the following, we will sketch how these three steps are practically carried out in the two agent platforms, JADE and JaCaMo.

5.1. Programming JADE Agents with 2COMM

Broadly speaking, a JADE agent can be seen as a set of behaviors that can be executed when needed. In this sense, a behavior (or a set of behaviors) represents a strategy to achieve one of the agent's goals. In order to use 2COMM, a JADE agent has to perceive the 2COMM workspace. Observing the workspace, however, does not require the agent to proactively sense it because agents can register to the commitment artifacts of interest and then will be notified by these artifacts whenever a social event occurs. Agent programmers, thus, have to implement the behaviors either by adopting the *Entice* schema and then taking the initiative of creating a commitment or by adopting the *Cooperate* schema and then reacting to the social events that are relevant to their agents; that is, the social events the agent at hand is expected to handle.

The *Entice* schema is implemented by a behavior that, inside its *action()* method, includes the execution of a `create()` operation on some commitment artifacts. The effect of such an operation is to bring the commitment state from `null` to `active` (specifically, `conditional`). Of course, this change of state is automatically notified by 2COMM to all the agents that have focused on that commitment artifact. As concerns the *Cooperate* schema, the agent programmer implements a behavior reacting to each event of interest by exploiting the *handleEvent()* method that allows the programmer to add one (or more) behavior(s) to the agent depending on the event intercepted from 2COMM, and on which of the two sub-cases of the *Cooperate* schema is needed. For instance, if the social event is the detachment of a commitment involving the agent as debtor, *handleEvent()* activates a behavior that will satisfy the very same commitment. The behavior body could encompass operations upon the environment (which, in turn, will make commitments evolve) or could also involve the creation of additional commitments. Conversely, if the social event is the creation of a commitment involving the agent as creditor, *handleEvent()* triggers a behavior designed to detach such a commitment, being the agent interested in the consequent condition. Summing up, by means of *handleEvent()*, the occurrence of a social event relevant for the agent activates a corresponding behavior, properly implemented by the agent programmer.

The following pseudo-code outlines the implementation of a JADE agent that follows the *Entice* and *Cooperate* schema.

```
1  public class MyBehavior extends SomeJadeBehavior implements CommitmentObserver {
2   public void action() {
3    ArtifactId art = Role.createArtifact(myCommitmentArtifactName,
4                  MyCommitmentArtifactC1.class);
5    myRole = (SomeRole) (Role.enact(MyArtifact.ROLE_NAME, art,
6           new JadeBehaviorPlayer(this, myAgent.getAID())));
7    myRole.startObserving(this);
8    // Entice schema: add behavior(s) to create commitment C1
9    // ECA: ON goal_to_achieve IF local_condition THEN create(C1)
10   myAgent.addBehavior(new MyEnticeBehaviorForCommitmentC1());
11   // Entice schema: add behavior(s) to create commitment C2
12   ...
13  }
14  public void handleEvent(SocialEvent e, Object... args) {
15   if (e.getElementChanged().getElType() == SocialStateElementType.COMMITMENT) {
```

```
16      Commitment c = (Commitment) e.getElementChanged();
17      // Cooperate schema as debtor of c
18      // ECA: ON detacted(c) IF local_condition DO consequent.
19      if(c.getDebtor().equals(myRole) && c.getConsequent().equals(...) &&
20        c.getLifeCycleStatus() == DETACHED) {
21          // add behavior(s) to satisfy c
22        myAgent.addBehavior(new MySatisfyBehavour());
23      }
24      // Cooperate schema as creditor of c
25      // ECA: ON create(c) ∨ conditional(c) IF consequent interesting DO antecedent.
26      \medskip
27      else if(c.getCreditor().equals(myRole) && c.getAntecedent().equals(...) &&
28         c.getLifeCycleStatus() == CONDITIONAL) {
29        // add behavior(s) to detach c
30        myAgent.addBehavior(new MyDetachBehavour());
31      }
32    }
33  }
34  ...
35 }
```

Summing up, the basic schema for implementing a JADE agent using social commitments requires two steps. First, apply the *Entice* schema within the agent method *action()* by allocating a commitment artifact (Line 3) and then adding to the agent a behavior for creating the commitment in such an artifact (Line 10). The *Entice* schema is therefore replicated for each commitment the agent intends to create. Second, apply the *Cooperate* schema within the method *handleEvent()*: for each relevant event `e` intercepted by *handleEvent()* and related to a specific commitment `c`, implement either the debtor or the creditor case depending on the role played by the agent in `c`.

5.2. Programming JaCaMo Agents with 2COMM

The JaCaMo framework adopts Jason as the agent programming language. Jason is implemented in Java and extends the agent programming language AgentSpeak(L). Jason agents have a BDI architecture including a belief base and a plan library storing the set of plans available to the agent for execution. Furthermore, it is possible to specify achievement (operator "!") and test (operator "?") goals. A plan has the structure $triggering\_event : \langle context \rangle \leftarrow \langle body \rangle$. *triggering_event* denotes the event the plan handles (a belief/goal addition or deletion), while *context* specifies the circumstances in which the plan could be used, and *body* expresses the course of action that should be taken. In JaCaMo, Jason agents can natively perceive and act upon artifacts programmed in CArtAgO. More precisely, when an agent *focuses* on an artifact, the artifact's observable properties are automatically mapped to beliefs in the agent's belief base. At the same time, in the plans' bodies, operations on artifacts can be specified. Commitment artifacts allow the agent to perceive commitment by representing them as observable properties and to act upon it by providing standard operations for their manipulation.

During execution, the agent infrastructure performs the sense-plan-act cycle that allows agents to evaluate which plans can be triggered for execution each time an event occurs. In this setting, social events generated by commitment artifacts during an interaction can be modeled as regular Jason events, mapped from the artifacts' observable properties to the agent beliefs. The adoption of commitment artifacts that notify the occurrence of social events to focusing agents allows plan specifications whose triggering events involve social events (e.g., commitment creation). For example, this is the case that deals with commitment addition:

$$+cc(Debtor, Creditor, Antecedent, Consequent, Status) :$$
$$\langle context \rangle \leftarrow \langle body \rangle.$$

The plan is triggered when a commitment that unifies with the plan head appears in the social state with the specified status. Contextual conditions must hold, as well. The syntax is the standard for Jason plans. *Debtor* and *Creditor* are to be substituted by the proper role identifiers.

The *Entice* and *Cooperate* schemas are easily implemented by defining a set of Jason plans. In particular, the *Entice* schema is implemented by a plan whose body includes the creation of a specific commitment (this requires the invocation of a `create` operation on the right commitment artifact). The *Cooperate* schema, in turn, is realized through the definition of plans whose triggering events match with the social events that are relevant for the agent. In particular, an agent should be equipped with at least one plan to deal with the detachment of the commitments involving the agent itself as debtor. The plan body is the expected to include the operations over the environment needed to lead the commitment to satisfaction. Note that this does not mean that the agent is required to directly perform the actions causing the consequent condition to hold. The agent could, in fact, rely on the collaboration with another agent. The following pseudo-code shows a prototype implementation for a JaCaMo agent:

```
// Entice schema: create a commitment c to get a goal
// ECA: ON goal_to_achieve IF local_condition THEN create(C1)
+!goal
   : context
 <- createMyCommitment. // Execute create operation on commitment artifact

// Cooperate schema as debtor: plan reacting to the detachment of a commitment
// ECA: ON detacted(c) IF local_condition DO consequent.
+cc(My_Role_Id, Creditor, Antecedent, Consequent, "DETACHED")
   : enactment_id(My_Role_Id) & ...
 <- ... plan to achieve Consequent ...

// Cooperate schema as creditor: plan reacting to commitment creation
// ECA: ON create(c) ∨ conditional(c) IF consequent interesting DO antecedent.
+cc(Debtor, My_Role_Id, Antecedent, Consequent, "CONDITIONAL")
   : enactment_id(My_Role_Id) & ...
 <- ... plan to achieve Antecedent ...
...
```

6. Implementing the Logistics Scenario

The explicit representation of social relationships by means of social commitments, reified in the environment as resources made accessible to the agents, provides a programming schema and an execution infrastructure for building agents with a uniform implementation of social behaviors. On this foundation, agents can be developed by programming how they should manipulate these relationships (e.g., create them) and react to relevant changes in them. In this section, we show how the conceptual schema presented above can be applied to the use case inspired from the logistics planning domain introduced in Section 3 to support the realization of interaction between agents in a distributed setting. We discuss two implementations, one in JADE and one in JaCaMo, both relying on 2COMM for the realization of the interaction layer, to show the generality and uniformity of the approach.

6.1. Modeling the Environment

Recalling Section 3, the interaction between the different agents in the scenario can be modeled by the following set of commitments:

$$\begin{aligned}
&\boldsymbol{c_1} : \mathsf{C}(seller,\ customer,\ pay(500, seller),\ at(goods, D))\\
&\boldsymbol{c_2} : \mathsf{C}(trk1,\ seller,\ pay(50, trk1) \wedge at(goods, A),\ at(goods, B))\\
&\boldsymbol{c_3} : \mathsf{C}(pln1,\ seller,\ pay(200, pln1) \wedge at(goods, B),\ at(goods, D))\\
&\boldsymbol{c_4} : \mathsf{C}(trk3,\ pln1,\ pay(50, trk3) \wedge at(goods, C),\ at(goods, D))
\end{aligned}$$

For each commitment, a corresponding commitment artifact, embodied in the environment, is defined in 2COMM. The following piece of code shows an excerpt of the artifact that maintains c_1:

```
public class C1 extends CommitmentArtifact {

 private RoleId debtor;
 private RoleId creditor;
 private Commitment c;

 static {
  addEnabledRole("debtorC1", DebtorC1.class);
  addEnabledRole("debtorC1", CreditorC1.class);
 }

 // Creditor and debtor role enactment methods for JADE and JaCaMo
 @OPERATION
 protected void enact(String roleName, IPlayer p, OpFeedbackParam<Role> r) { ... }
 @OPERATION
 protected void enact(String roleName) { ... }

 // Commitment manipulation operations
 @OPERATION public void createC1() {
  RoleId agent = getRoleIdByRoleName(getCurrentOpAgentId().getAgentName());
  if (!agent.equals(debtor)) {
   failed("Only the debtor can create a commitment");
  }
  c = new Commitment(debtor, creditor,
     new Fact("pay", "500", debtor.toString()),new Fact("at", "pkg", "D"))
  createCommitment(c);
 }
 @OPERATION public void releaseC1() {
  RoleId agent = getRoleIdByRoleName(getCurrentOpAgentId().getAgentName());
  if (!agent.equals(creditor)) {
   failed("Only the creditor can release a commitment");
  }
  releaseCommitment(c);
 }
 @OPERATION public void cancelC1() {
  RoleId agent = getRoleIdByRoleName(getCurrentOpAgentId().getAgentName());
  if (!agent.equals(debtor)) {
    failed("Only the debtor can cancel a commitment");
  }
  cancelCommitment(c);
 }

 //Fact assertion operations
 @OPERATION public void assertPay(String pred, String who) {
  assertFact(new Fact("pay", pred, who));
 }
 @OPERATION public void assertAt(String what, String where) {
  assertFact(new Fact("at", what, where));
 }

 // Debtor role class
 public class DebtorC1 extends CommitmentRole {
  @RoleAction public void createC1() {
   try {
     doAction(this.getArtifactId(), new Op("createC1"));
   } catch (ActionFailedException | CartagoException e) {...}
  }
  @RoleAction public void cancelC1() {...}
  ...
 }

 // Creditor role class
 public class CreditorC1 extends CommitmentRole {
```

```
64    @RoleAction public void releaseC1() {
65     try {
66       doAction(this.getArtifactId(), new Op("releaseC1"));
67     } catch (ActionFailedException | CartagoException e) {...}
68    }
69    ...
70   }
71   ...
72  }
```

The commitment artifact includes enact methods for JADE and JaCaMo agents (omitted for simplicity), which allow the agents to adopt the debtor and creditor roles of the commitment. Role enactment enables the execution of the commitment operations made available by the artifact. In particular, the artifact provides two debtor-dedicated operations (see Lines 19 and 35) to create and cancel the commitment and one creditor-dedicated operation to release it (see Line 28). When such operations are executed, the 2COMM infrastructure takes care of updating the commitment state and the artifact observable state. Similarly, agents observing the artifact are notified about the occurrence of the social event resulting from the execution of the operation (e.g., the commitment creation). As inner classes, at Lines 52 and 63, the artifact provides two commitment roles amounting to the debtor and creditor, through which, upon enactment, the agents can execute the allowed operations on the artifact. Finally, the artifact provides two operations to assert facts that are relevant for the commitment progression. Intuitively, a fact records the occurrence of a (social) event. Generally speaking, this type of operation is used by the 2COMM infrastructure for propagating the effects of operations on the environment to all the existing (commitment) artifacts. When a fact is asserted into a commitment artifact, the infrastructure automatically updates the commitment state, if needed. In this example, the two facts that can be asserted trace a payment and the delivery of goods at a specific location. The artifacts reifying c_2, c_3, and c_4 are similar to the one presented above, and thus, they are not discussed in further detail.

For the sake of completeness, we exemplify also a standard CArtAgO artifact used to model part of the environment. In particular, we sketch the `Logistics` artifact that keeps track of the current position of the shipped goods and provides to the agents the operations to pay a given amount of money to another agent and to move the goods from one location to another one. Events resulting from the execution of such operations are then propagated to commitment artifacts trough the assertion of the corresponding facts.

```
1  public class Logistics extends Artifact {
2
3   private String packageName;
4   private Location currentLocation;
5
6   private enum Location { A, B, C, D }
7
8   static { addEnabledRole("logisticRole", LogisticRole.class); }
9
10  @OPERATION public void init(String packageName) {
11   this.packageName = packageName;
12   defineObsProperty("packageName", packageName);
13  }
14  @OPERATION public void pay(int what, String who) {
15   defineObsProperty("pay", what, who);
16   setChanged();
17   notifyObservers(...);
18  }
19  @OPERATION public void move(String pkg, String where) {
20   if (!packageName.equals(pkg)) {
21    failed("Wrong package name");
22   } else if (where.equals("A")) {
23    currentLocation = Location.A;
```

```
    defineObsProperty("currentLocation", Location.A.toString());
    setChanged();
    notifyObservers(...);
   } else if (where.equals("B")) { ... }
   ...
  }
  ...
}
```

6.2. Programming the Agents

Let us now focus on the programming of agents to see how the schema presented in Section 4 can be applied to develop JADE and JaCaMo agents in a uniform way.

6.2.1. JADE Agents

As discussed above, the implementation of a JADE agent driven by the social relationships requires implementing an agent behavior where to the *action()* and *handleEvent()* methods are applied the *Entice* and *Cooperate* schemas, respectively. As an illustration, the following piece of code shows an excerpt of agent *trk*1, involved as debtor in c_2.

```
public class Trk1 extends Agent {

 DebtorC2 roleC2;
 LogisticRole logRole;

 protected void setup() { addBehavior(new Trk1Behavior()); }

 public class Trk1Behavior {
  @Override public void action() {
        // Entice schema:
        // ECA: ON need_money IF can_deliver_from_A_to_B
        //   DO create (c2 : C(trk1, seller, pay(50, trk1) ∧ at(goods, A), at(goods, B)))
   ArtifactId log = Role.createArtifact("log", Logistics.class);
   ArtifactId c2 = Role.createArtifact("c2", C2.class);
   roleC2 = (DebtorC2) (Role.enact("debtorC2", c2,
       new JadeBehaviorPlayer(new Behavior[]{this}, myAgent.getAID())));
   roleC2.startObserving(this);
   logRole = (LogisticRole) (Role.enact("logRole", log,
       new JadeBehaviorPlayer(new Behavior[]{this}, myAgent.getAID())));
   logRole.startObserving(this);
   roleC2.createC2();
  }
  @Override public void handleEvent(SocialEvent e, Object... args) {
        // Cooperate schema as debtor:
        // ECA: ON detached(c2) IF true DO deliver_goods_at_B
   if(e.getElementChanged().getElType() == SocialStateElementType.COMMITMENT) {
     Commitment c = (Commitment) e.getElementChanged();
     if(c.getDebtor().toString().equals(roleC2.toString()) &&
      c.getAntecedent().equals(new CompositeExpression(LogicalOperatorType.AND,
         new Fact("at","goods","A"),
         new Fact("pay", "50", c.getDebtor().toString()))) &&
      c.getConsequent().equals(new Fact("at", "goods", "B")) &&
      c.getLifeCycleStatus() == LifeCycleState.DETACHED) {
      myAgent.addBehavior(iSatisfyC2());
     }
   }
  }

  @Override public Behavior iSatisfyC2() { return new SatisfyC2(); }

  public class SatisfyC2 extends OneShotBehavior {
```

```
42      @Override public void action() { logRole.move("goods", "B"); }
43    }
44  }
45 }
```

The agent has a behavior `Trk1Behavior` that is activated as soon as the agent starts the execution. After having adopted the debtor role in the artifact managing c_3 (Line 16), the agent immediately creates the commitment (Line 21). This part of the behavior realizes the *Entice* schema. The same behavior also implements a `handleEvent` method that realizes the *Cooperate* schema (see Line 23). In particular, the method handles the social event arisen from the detachment of the previously-created commitment. In that case, a new behavior is added to the agent's behavior repository (Line 34). The behavior includes the execution of the actions needed to bring the commitment to satisfaction. In this case, the agent simply has to move the goods to B (Line 42). The two behaviors described above completely characterize the $trk1$ agent.

The other agents have the same structure, and their code, for brevity, is not reported here. However, it is possible to make some considerations. The *customer* agent, being only a creditor of one commitment, does not have any behavior implementing the entice schema. Instead, if interested in the consequent condition of c_1, the agent's designer could implement the cooperate schema by defining a suitable *handleEvent()* to detach it, once created by *seller*. In this case, the method would amount to the activation of a behavior leading to the payment of 500 Euros to the seller. It is interesting to point out that the entice schema could be triggered by the occurrence of a social event, as well. This holds, for instance, for the creation of c_3 by $pln1$ and of c_1 by *seller*. Since the plane has to rely on the collaboration of $trk3$ to bring about the consequent condition of c_3, it is reasonable to assume that it will create its commitment only after the creation of c_4. Instead, $pln1$'s behavior implementing the cooperate schema for c_3 will include both the movement of the goods from B to C and the detachment of c_4 trough a payment of 50 Euros to $trk3$ for the final part. Similarly, *seller* will create c_1 only as a consequence of the creation of c_2 and c_3. The shape of these commitments creates a chain of cooperation between the agents. In order to satisfy c_1, once detached, *seller* has to act, in turn, to detach c_2 and c_3. In the same way, in order to satisfy c_3, $pln1$ has to detach c_4. Only if every agent properly handles the social events related to its commitments, thereby following the two proposed schemas, the interaction will succeed, and the customer will receive the goods for which it has paid.

6.2.2. JaCaMo Agents

As shown in Section 5.2, the *Entice* and *Cooperate* schemas can be implemented in a Jason agent by defining a set of plans that, respectively, create commitments and react to social events. The following piece of code shows an excerpt of the $trk1$ agent, implemented in Jason:

```
1  !start.
2  // Entice schema:
3  // ECA: ON need_money IF can_deliver_from_A_to_B
4  //   DO create (c2 : C(trk1, seller, pay(50, trk1) ∧ at(goods, A), at(goods, B)))
5  +!start
6     : focused(_,c2,C2ArtId) &
7   <- enact("debtorC2")[artifact_id(C2ArtId)];
8      createC2.
9  //Cooperate schema as debtor:
10 // ECA: ON detached(c2) IF true DO deliver_goods_at_B
11 +cc(MyId,_,"(at(goods,A) AND pay(50,trk1))","at(pkg,B)","DETACHED")
12    : focused(_,log,LogArtId) &
13  <- move("goods","B")[artifact_id(LogArtId)].
```

The first plan (Line 5) is triggered as soon the execution starts. This plan implements the entice schema for c_2. After enacting the debtor role in the artifact, the agent creates the commitment representing its offer. The second plan (Line 11) is an example of the *Cooperate* schema in the debtor case; it is in fact triggered when commitment c_2 is detached. The body of the plan, just like the

analogous behavior in JADE, contains the operations to be performed over the environment to satisfy the detached commitment. In this case, the only operation to be performed is to move the goods to B.

As an example of the *Cooperate* schema in the creditor case, let us consider the *customer* agent that accepts an offer made by *seller* by detaching commitment c_1:

```
1 //Cooperate schema as creditor:
2 // ECA: ON conditional(c1) IF have_goods_at_D DO pay_for_goods
3 +cc(_,MyId,"pay500(s)","at(goods, D)","CONDITIONAL")
4   : enactment_id(MyId,c1,C1ArtId)
5  <- pay(500,"seller")[artifact_id(C1ArtId)].
```

Of course, whenever an agent is involved in a commitment, either as debtor or creditor, the programmer has to pay attention in implementing at least the plans (or behaviors) that comply with the two proposed schemas. For instance, *seller* is involved in many commitments (i.e., c_1, c_2, and c_3), and hence, the programmer has to implement the plans that handle the progression of each of these commitments, by applying the *Entice* or *Cooperate* schema as necessary.

6.3. A More Complex Scenario

An interaction devised in terms of social relationships explicitly created among the different actors enables a form of flexibility that neither message-based protocols, nor interaction components can obtain. These two approaches are basically prescriptive: they describe a specific sequence of steps through which an interaction can correctly evolve. In real-world cases, however, there exist alternative paths that could bring the same result. Prescriptive approaches either ignore these alternatives or encode them by making the resulting system potentially very complex or rigid. Interactions defined in terms of sets of social commitments that can be created and manipulated by agents do not impose any strict ordering of the messages (or actions). It follows that agents generally have greater flexibility in deciding how to operate.

As an illustration, let us consider a scenario in which the *customer* agent is not happy with the offer made by the *seller* with the creation of c_1. More in detail, let us assume that the customer is willing to pay at most 450 Euros for the goods and that it would like to pay only after the delivery. It could, then, make a counteroffer to *seller* through the creation of an additional commitment. Additionally, *customer* could decide to release c_1 (despite not being obliged to do so), to make clear that the original offer made by *seller* is unacceptable for it. In order to still get a good return from the sale, *seller* could try to negotiate with the couriers to have better prices for the shipments. Upon the negotiation phase, *seller* could decide to accept or reject the counteroffer. For instance, it could negotiate with truck $trk2$ the transportation of the goods from A to B for only 20 Euros (instead of 50, as asked by $trk2$). Similarly, *seller* could decide to rely on the plane only for the route from B to C (provided that $pln1$ offers that kind of service, as well) and then negotiate a cheaper price for $B - C$ with $trk4$. If these negotiations will be concluded positively, a new set of commitments will be created as shown in the following:

$$\begin{aligned}
&\boldsymbol{c_5} : \mathsf{C}(customer,\ seller,\ at(pkg, D),\ pay(450, seller))\\
&\boldsymbol{c_6} : \mathsf{C}(seller,\ trk2,\ c_7,\ pay(20, trk2))\\
&\boldsymbol{c_7} : \mathsf{C}(trk2,\ seller,\ pay(20, trk2) \wedge at(goods, A),\ at(goods, B))\\
&\boldsymbol{c_8} : \mathsf{C}(pln1,\ seller,\ pay(100, pln1) \wedge at(goods, B),\ at(goods, C))\\
&\boldsymbol{c_9} : \mathsf{C}(seller,\ trk2,\ c_{10},\ pay(50, trk4))\\
&\boldsymbol{c_{10}} : \mathsf{C}(trk4,\ seller,\ pay(50, trk4) \wedge at(goods, C),\ at(goods, D))
\end{aligned}$$

c_5 encodes the counteroffer made by *customer* to *seller*. c_6 and c_9 represent the offers made by the seller to $trk2$ and $trk4$ related to $A - B$ and $C - D$. These two commitments contain, as antecedent conditions, two additional commitments: c_7 and c_{10}, respectively. The meaning of such notation is that the creation of c_7 and c_{10} leads to the detachment of c_6 and c_9, respectively. Should $trk2$ decide that the offer made by *seller* is acceptable, it could create c_7 committing to offer its services at the agreed prices. The *seller* would then be asked to pay, lest it violates c_6. The same holds for $trk4$. c_8, finally,

encodes the second type of service offered by $pln1$, namely the ability to bring the goods only from B to C upon payment of 100 Euros.

Depending on which commitments in the set $\{c_1, \ldots, c_{10}\}$ are created and on how they progress (i.e., they could be detached, released, or even canceled), the interaction takes very different shapes. For instance, $trk2$ and $trk4$ could decide to refuse (or ignore) the offer made by *seller*, forcing it to refuse the counteroffer made by the customer. The customer, as a consequence, could then decide to accept the original offer or to give up. Still, the *seller* could decide to accept the counteroffer and then still rely on $trk1$ and $pln1$ for the shipment, thereby making a lower profit, but making the *customer* happy. A commitment-based representation of social relationships, reified in the environment as resources, provides a uniform and comprehensive representation of all these alternative paths. Furthermore, the *Entice* and *Cooperate* schemas provide a structured guideline to develop agents able to properly manipulate commitments and to react to the corresponding relevant social events.

6.4. Remarks

2COMM provides JADE and JaCaMo agents with an explicit representation of social relationships (i.e., commitments) that agents can directly manipulate. We have seen that a first positive consequence is that agents can be programmed by following a uniform schema that is agent-platform independent. This is not, however, the unique advantage that one gains when 2COMM is integrated within an agent platform. Table 1 synthesizes a comparison between JADE and JaCaMo, highlighting aspects that are improved or added by an explicit representation of social relationships as commitments.

Table 1. Comparison among JADE, JaCaMo, and improvements provided by an explicit representation of social relationships.

	JADE	**JADE + 2COMM**	**JaCaMo**	**JaCaMo + 2COMM**
Programmable interaction means	X	✓	✓	✓
Notification of social relationships of interaction	X	✓	X	✓
Interaction/agent logic decoupling	X	✓	X	✓
Expected behaviors reasoning	X	✓	X	✓
Definition of structured schemas for interaction	✓	✓	X	✓
Runtime interaction monitoring	X	✓	✓	✓
Agent programming aware of social relationships	X	✓	X	✓

Programmable interaction means: 2COMM reifies social commitments as artifacts embodied in the environment, in particular as *commitment artifacts*. This significantly improves JADE, where programmable interaction means are missing. On the other hand, JaCaMo already integrates the CArtAgO component and, hence, already includes programmable artifacts that can be used to mediate the interaction among the agents.

Notification of social relationships of interaction: Commitment artifacts specialize CArtAgO artifacts because they implement the commitment lifecycle. Any change of state in the artifact, thus, corresponds to a change in the interaction state. For this reason, we can state that 2COMM enables a form of automatic notification about social relationships that is not provided either by JADE or by JaCaMo.

Interaction/agent logic decoupling: In JADE and JaCaMo, no specific abstractions are used for modeling and implementing the interaction logic. In JADE, agents interact by exchanging messages. In JaCaMo, agents can modify an artifact for communicating with others. In both cases, however, the semantics of the message or of the artifact operation must be encoded within the interacting agents. In 2COMM, instead, the semantics of the interaction is given in terms of commitments. Therefore, when an agent acts upon the environment (e.g., pays for some goods), the meaning, and implications, of such an action is modeled via a state change occurring in a specific commitment. It follows that the logic of

the interaction is not spread inside the agent code, but within the commitment artifacts. The 2COMM agents need just to possess the proper behaviors/plans for handling the commitment lifecycle.

Expected behaviors reasoning: As we have said, commitments have a normative power since they allow an agent to create an expectation about the behavior of others. The debtor of a commitment is expected by the creditor to bring about the consequent condition when the commitment will be detached. The debtor, in turns, creates a commitment because it anticipates that the creditor will be interested in the consequent condition and hence will be moved to detach the commitment. This mechanism of expectations is at the basis of the general schema of agent programming that we have proposed. Neither JADE nor JaCaMo can rely upon an analogous mechanism.

Definition of structured schemas for interaction: As pointed out above, expectations allow us to define a general schema for programming agents that is independent of the agent-platform since it only depends on the lifecycle of commitments.

Runtime interaction monitoring: Although we have not discussed this aspect in detail, 2COMM facilitates the implementation of monitors that supervise the whole interaction occurring in the system. It would be sufficient, in fact, to have an agent monitoring each commitment artifact of a workspace and hence capturing every change of state in the commitments. Of course, this is a direct consequence of having social relationships explicitly modeled as first-class elements and not implicitly modeled via message passing as in JADE. In JaCaMo, a monitor could also be implemented as an observer of the artifacts in the environment.

Agent programming aware of social relationships: The proposed programming schema guides a programmer in developing an agent that is aware and compliant with its engagements. In particular, the *Cooperate* schema helps a programmer identify what social events are relevant for the agent being implement and what is expected of the agent given its engagements.

7. Discussion and Conclusions

In this work, we have presented a general schema for programming socially-responsive agents. We have argued how, when agents can directly manipulate their engagements via commitments, it is possible to gain some software engineering advantages, such as modularity and flexibility. We have demonstrated this in practice by showing how agents can be programmed uniformly in two substantially different platforms: JADE and JaCaMo. The core of the proposal relies on the 2COMM infrastructure, which allows agents to interact by following an accepted set of regulations, with a self-governance approach. Self-governance mechanisms rely on the reification of commitments. These are, in all respects, resources that are made available to stakeholders and that are realized by means of artifacts. The proposal is characterized, on the one hand, by the flexibility and the openness that are typical of MAS and, on the other, by the modularity and the compositionality that are typical requirements of the methodologies for design and development. One of the strong points of the proposal is the decoupling between the design of the agents and the design of the interaction, which builds on the decoupling between computation and coordination done by coordination models, like tuple spaces. This is a difference with respect to JADE or JaCaMo where no decoupling occurs: a schema of interaction is projected into a set of JADE behaviors or Jason plans, one for each role. Binding the interaction to ad hoc behaviors/plans does not allow having a global view of the protocol and complicates its maintenance.

2COMM supports programming JADE and Jason agents, by following a uniform approach. Agents' programming can leverage the general schema we have presented here, which complements the methodology introduced in [36]. It also simplifies the interaction of different agent platforms thanks to its connectors, which make protocol artifacts accessible and, thus, allow mediated communication between agents that belong to different platforms. This feature fulfills the purpose of supporting the development of heterogeneous and open agent systems. Therefore, for instance, 2COMM enables the interaction of JADE agents with JaCaMo agents in a transparent and seamless way: it is not necessary to adapt an agent implementation to the platforms on which the other agents of the system run. As

concerns implementation, connectors bridge between CArtAgO and the used agent platforms. Any agent can take part in the interaction sessions with others simply by using a commitment artifact. Note that Jason (which is part of JaCaMo) allows changing the communication infrastructure, switching to that of JADE; however, this choice is to be done a priori and has an impact on the design. 2COMM does not impose any choice a priori, guaranteeing the interaction between any pair of agent platforms for which a 2COMM connector exists.

2COMM complements the obligation-based specification of organizations, specifically suiting those situations where interaction is not subject to an organizational guideline, like in the case when interaction is among agents and each agent decides what is best for itself or when guidelines amount to declarative, under-specified constraints that still leave agents the freedom to make strategic decisions on their behavior. In this case, interaction strongly relies on the two basic notions of goal and of engagement. For a thorough discussion of the differences between our proposal and organizational or normative approaches, please check [37]. Testing 2COMM with Jason and JADE proved that programming agents starting from their desired interaction can be a valuable starting point, which can be extended towards a methodology useful for open and heterogeneous scenarios. We intend to explore this direction by adding connectors for different agent platforms. The implementation of a scenario like the logistic one serves the purpose of showing the validity and suitability of the approach from a conceptual standpoint. Indeed, it provides a proof of concept to highlight the benefits coming from the development of socially-responsive agents in a realistic use case. On the other hand, the scalability of the approach strongly depends on the underlying infrastructure. We have shown how a relational representation of interaction induces a programming schema that can support the development of a system encompassing multiple agents. Nonetheless, in order to be used in any real-world scenario, it is mandatory that the underlying programming infrastructure is carefully engineered with particular attention on the features of the given application domain.

Recently, we developed on top of 2COMM a commitment-based typing system [38] for JADE agents. Such typing includes a notion of compatibility, based on subtyping, which allows for the safe substitution of agents into roles along an interaction that is ruled by a commitment-based protocol. Type checking can be done dynamically when an agent enacts a role.

Although this is outside the scope of this paper, the reification of social commitments as resources in the environment paves the way for the realization of a runtime interaction monitoring system, as already pointed out in the previous section. In this sense, the conceptual architecture outlined in this paper could serve as a basis for the identification of undesirable interaction patterns (e.g., an agent with conflicting commitments). A possible extension of this work could involve the definition of proper metrics to be included in a monitor, following the line drawn in [39].

As a final remark, it would be interesting to relate the proposal to works aimed at systematizing approaches to multiagent system modeling, like [40], which describes a meta-model for agent modeling approaches. To this aim, however, it would first be necessary to encompass into the meta-model also methodologies that are typical of commitment-based approaches, like [41,42].

Author Contributions: Conceptualization, M.B., C.B., R.M., and S.T.; methodology, M.B., C.B., R.M., and S.T.; software M.B., C.B., R.M., and S.T.; validation, M.B., C.B., R.M., and S.T.; writing–original draft preparation, M.B., C.B., R.M., and S.T.; writing–review and editing, M.B., C.B., R.M., and S.T.

Funding: This research was funded by Regione Piemonte grant number 320-30 "CANP - La Casa nel Parco".

Acknowledgments: The authors would like to thank Federico Capuzzimati for the discussions and contribution to the initial version of the 2COMM software.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

2COMM	Communication and Commitment
A&A	Agents and Artifacts
CArtAgO	Common ARTifact infrastructure for AGents Open environments
CNP	Contract Net Protocol
JaCaMo	Jason, CArtAgO, Moise
JADE	Java Agent DEvelopment framework
ECA	Event-Condition-Action
MAS	Multiagent System

References

1. Bordini, R.H.; Braubach, L.; Dastani, M.; Fallah-Seghrouchni, A.E.; Gomez-Sanz, J.J.; Leite, J.; O'Hare, G.M.P.; Pokahr, A.; Ricci, A. A Survey of Programming Languages and Platforms for Multi-Agent Systems. *Informatica (Slovenia)* **2006**, *30*, 33–44.
2. Bellifemine, F.; Bergenti, F.; Caire, G.; Poggi, A. JADE—A Java Agent Development Framework. In *Multi-Agent Programming: Languages, Platforms and Applications*; Springer: Berlin/Heidelberg, Germany, 2005; Volume 15, pp. 125–147.
3. Omicini, A.; Zambonelli, F. TuCSoN: A Coordination model for Mobile Information Agents. In Proceedings of the 1st International Workshop on Innovative Internet Information Systems (IIIS'98), Pisa, Italy, 8–9 June 1998; pp. 177–187.
4. Brazier, F.M.T.; Dunin-Keplicz, B.M.; Jennings, N.R.; Treur, J. Desire: Modelling Multi-Agent Systems in a Compositional Formal Framework. *Int. J. Coop. Inf. Syst.* **1997**, *06*, 67–94.
5. Boissier, O.; Bordini, R.H.; Hübner, J.F.; Ricci, A.; Santi, A. Multi-agent oriented programming with JaCaMo. *Sci. Comput. Program.* **2013**, *78*, 747–761.
6. Singh, M.P. *A Social Semantics for Agent Communication Languages*; Issues in Agent Communication; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1916, pp. 31–45.
7. Baldoni, M.; Baroglio, C.; Capuzzimati, F. Programming JADE and Jason Agents Based on Social Relationships Using a Uniform Approach. In *Advances in Social Computing and Multiagent Systems*; Koch, F., Guttmann, C., Busquets, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; Volume 541, pp. 167–184.
8. Baldoni, M.; Baroglio, C.; Capuzzimati, F. A Commitment-Based Infrastructure for Programming Socio-Technical Systems. *ACM Trans. Internet Technol.* **2014**, *14*, 23:1–23:23.
9. Singh, M.P. An Ontology for Commitments in Multiagent Systems. *Artif. Intell. Law* **1999**, *7*, 97–113.
10. Conte, R.; Castelfranchi, C.; Dignum, F. Autonomous Norm Acceptance. In *Intelligent Agents V, Agent Theories, Architectures, and Languages*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1998; Volume 1555, pp. 99–112.
11. Weyns, D.; Omicini, A.; Odell, J. Environment as a first class abstraction in multiagent systems. *Auton. Agents Multi-Agent Syst.* **2007**, *14*, 5–30.
12. Omicini, A.; Ricci, A.; Viroli, M. Artifacts in the A&A meta-model for multi-agent systems. *Auton. Agents Multi-Agent Syst.* **2008**, *17*, 432–456.
13. Ossowski, S. Coordination in Multi-Agent Systems: Towards a Technology of Agreement. In *Multiagent System Technologies*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 2–12.
14. Adam, C.; Gaudou, B. BDI agents in social simulations: A survey. *Knowl. Eng. Rev.* **2016**, *31*, 207–238.
15. Smith, R.G. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Trans. Comput.* **1980**, *29*, 1104–1113.
16. Philippsen, M. A survey of concurrent object-oriented languages. *Concurr. Pract. Exp.* **2000**, *12*, 917–980.
17. D'Inverno, M.; Luck, M.; Noriega, P.; Rodríguez-Aguilar, J.A.; Sierra, C. Communicating open systems. *Artif. Intell.* **2012**, *186*, 38–94.
18. Criado, N.; Argente, E.; Noriega, P.; Botti, V. Reasoning about constitutive norms in BDI agents. *Logic J. IGPL* **2014**, *22*, 66–93.
19. Esteva, M.; Rosell, B.; Rodríguez-Aguilar, J.A.; Arcos, J.L. AMELI: An Agent-Based Middleware for Electronic Institutions. In Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004), New York, NY, USA, 19–23 August 2004; pp. 236–243.

20. Arcos, J.L.; Noriega, P.; Rodríguez-Aguilar, J.A.; Sierra, C. E4MAS through Electronic Institutions. In *International Workshop on Environments for Multi-Agent Systems*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 184–202.
21. Hübner, J.F.; Sichman, J.S.; Boissier, O. S-MOISE$^+$: A Middleware for Developing Organised Multi-agent Systems. Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems. In Proceedings of the AAMAS 2005 International Workshops on Agents, Norms and Institutions for Regulated Multi-Agent Systems, ANIREM 2005, and Organizations in Multi-Agent Systems, OOOP 2005, Utrecht, The Netherlands, 25–26 July 2005; pp. 64–78.
22. Hübner, J.F.; Boissier, O.; Kitio, R.; Ricci, A. Instrumenting multi-agent organisations with organisational artifacts and agents: "Giving the organisational power back to the agents". *Auton. Agents Multi-Agent Syst.* **2009**, *20*, 369–400.
23. Baldoni, M.; Baroglio, C.; Capuzzimati, F.; Micalizio, R. Commitment-based Agent Interaction in JaCaMo+. *Fundam. Inf.* **2018**, *159*, 1–33.
24. Zatelli, M.R.; Ricci, A.; Hübner, J.F. Integrating interaction with agents, environment, and organisation in JaCaMo. *IJAOSE* **2016**, *5*, 266–302.
25. Boissier, O.; Bordini, R.H.; Hübner, J.F.; Ricci, A. Dimensions in programming multi-agent systems. *Knowl. Eng. Rev.* **2019**, *34*, e2.
26. Yolum, P.; Singh, M.P. Flexible protocol specification and execution: Applying event calculus planning using commitments. In Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, Bologna, Italy, 15–19 July 2002; pp. 527–534.
27. Winikoff, M.; Liu, W.; Harland, J. Enhancing Commitment Machines. In *International Workshop on Declarative Agent Languages and Technologies*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 198–220.
28. Yolum, P.; Singh, M.P. Designing and Executing Protocols Using the Event Calculus. In *Proceedings of the Fifth International Conference on Autonomous Agents*; ACM: New York, NY, USA, 2001; pp. 27–28.
29. Fornara, N.; Colombetti, M. Defining Interaction Protocols using a Commitment-based Agent Communication Language. In *Proceedings of the Second International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2003)*; ACM: New York, NY, USA, 2003; pp. 520–527.
30. Baldoni, M.; Baroglio, C.; Marengo, E.; Patti, V. Constitutive and Regulative Specifications of Commitment Protocols: a Decoupled Approach. *ACM Trans. Intell. Syst. Technol.* **2013**, *4*, 22:1–22:25.
31. Singh, M.P. Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition. In Proceedings of the Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, Melbourne, Australia, 14–18 July 2003; pp. 907–914.
32. Telang, P.R.; Singh, M.P. Specifying and Verifying Cross-Organizational Business Models: An Agent-Oriented Approach. *IEEE Trans. Serv. Comput.* **2012**, *5*, 305–318.
33. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. In *Programming Multi-Agent Systems in AgentSpeak Using Jason*; John Wiley & Sons: Chichester, West Sussex, England, 2007; Volume 8.
34. Ricci, A.; Piunti, M.; Viroli, M. Environment programming in multi-agent systems: An artifact-based perspective. *Auton. Agents Multi-Agent Syst.* **2011**, *23*, 158–192.
35. Hübner, J.F.; Boissier, O.; Kitio, R.; Ricci, A. Instrumenting multi-agent organisations with organisational artifacts and agents. *Auton. Agents Multi-Agent Syst.* **2010**, *20*, 369–400.
36. Baldoni, M.; Baroglio, C.; Capuzzimati, F.; Micalizio, R. Empowering Agent Coordination with Social Engagement. In *Congress of the Italian Association for Artificial Intelligence*; Springer: Berlin/Heidelberg, Germany, 2015; Volume 6934, pp. 89–101.
37. Baldoni, M.; Baroglio, C.; Capuzzimati, F.; Micalizio, R. Leveraging Commitments and Goals in Agent Interaction. In Proceedings of the 30th Italian Conference on Computational Logic, Genova, Italy, 1–3 July 2015; pp. 85–100.
38. Baldoni, M.; Baroglio, C.; Capuzzimati, F.; Micalizio, R. Type Checking for Protocol Role Enactments via Commitments. *J. Auton. Agents Multi-Agent Syst.* **2018**, *32*, 349–386.
39. Gutiérrez, C.; García-Magariño, I.; Fuentes-Fernández, R. Detection of Undesirable Communication Patterns in Multi-agent Systems. *Eng. Appl. Artif. Intell.* **2011**, *24*, 103–116.
40. García-Magariño, I. Towards the integration of the agent-oriented modeling diversity with a powertype-based language. *Comput. Stand. Interfaces* **2014**, *36*, 941–952.

41. Desai, N.; Chopra, A.K.; Singh, M.P. Amoeba: A Methodology for Modeling and Evolving Cross-organizational Business Processes. *ACM Trans. Softw. Eng. Methodol.* **2009**, *19*, 6:1–6:45.
42. Baldoni, M.; Baroglio, C.; Marengo, E.; Patti, V.; Capuzzimati, F. Engineering commitment-based business protocols with 2CL methodology. *J. Auton. Agents Multi-Agent Syst.* **2014**, *28*, 519–557.